

CSE 140L

Lab Assignment 5

Sequential Circuit Design

Due December 3, 2017, 11:59 pm

I. Introduction

This lab assignment consists of four parts. In the first part, you are asked to design your very own D, T, SR, and JK flip-flops in LogicWorks. In the second part of the lab, you will design and implement in behavioral Verilog a Mealy state machine controller for a vending machine. In the third part, you will modify your Booth's multiplier datapath from Lab 2 to incorporate a hardwired controller that will reduce the number of clock cycles needed for the multiplication operation. Finally, in the fourth part, you will duplicate and minimize a pattern recognition Mealy machine and solve for the minimized next state logic.

It is highly recommended that you review Chapters 11 and 13 to 15 on "Sequential Logic" in your textbook before starting this lab assignment. We provide you a brief overview below of some of the relevant concepts outlined in your book before proceeding with an explanation of the actual lab assignment.

II. Theory

Flip-Flop Construction

Before we can even start with designing FSM circuits, we need to figure out how to construct storage elements. Interestingly, this can be accomplished with combinational gates. So far, all logic involving combinational gates that you have implemented in the previous labs is composed of unidirectional datapaths. Data propagates directly from the primary inputs towards the primary output in a straight shot. This model of combinational logic, however, is rather constrained, as it does not directly allow for feedback to be taken into account when computing the primary output. This notion of feedback is quite powerful, and it is ultimately what allows us to construct storage elements.

The canonical storage element, termed an SR-latch, is shown below:

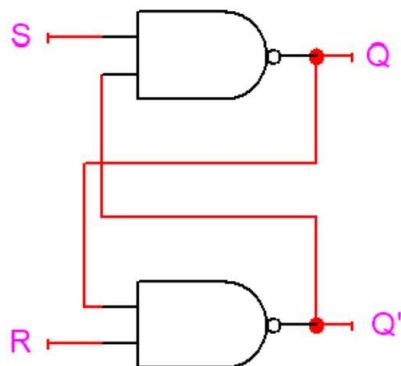


Figure 1: SR-latch

As you can see, the SR-latch is comprised of two NAND-gates and utilizes the output of each gate as feedback into the other. The behavior of the NAND-based SR-latch is summarized by the following table:

S	R	Q	Q'
0	0	1 (Invalid)	1 (Invalid)
0	1	1	0
1	0	0	1
1	1	Q	Q'

Table 1: NAND-based SR-latch Behavior

As you can see from the table, setting $S=0$ and $R=1$ will set Q to 1 and Q' to 0, whereas setting $S=1$ and $R=0$ will set Q to 0 and Q' to 1. If both S and R are set to 1, the previous values of Q and Q' are effectively held. Finally, setting $S=0$ and $R=0$ will cause Q and Q' to both be 1. The consequent violation of the property of Q and Q' being complements of each other is one of the reasons why this last case is deemed to be an invalid input combination.

As we can see, we are able to store values using only two combinational gates. In the previous labs, however, storage elements like the D flip-flop were synchronized with respect to the rising edge of a clock. How can we construct a D flip-flop with this sort of synchronization? The first step is to transform our SR-latch into a gated SR-latch, which is shown below:

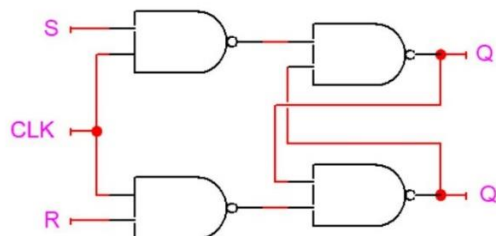


Figure 2: Gated SR-latch

There are two consequences to adding the additional NAND gates. First, the values from S and R will only propagate if the CLK input is high. If $CLK=0$, then the outputs of both of the new NAND gates are driven to logic 1, causing the original SR-latch to hold its value. Second, when $CLK=1$, the values from S and R are inverted when they propagate through the additional NAND gate. Thus, the input semantics are flipped compared to the non-gated version of the SR latch (and thus match the NOR based implementation outlined in your book). The new table for S and R in the gated SR latch is shown below:

S	R	Q (when CLK=1)	Q' (when CLK=1)
0	0	Q	Q'
0	1	0	1
1	0	1	0
1	1	1 (invalid)	1 (invalid)

Table 2: Gated SR-latch Behavior

One can then convert this SR-latch into a D-latch, as shown in the following figure:

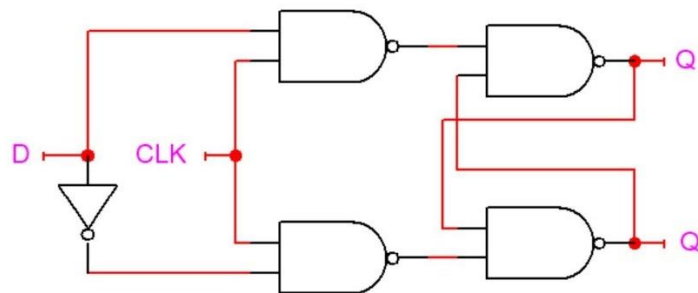


Figure 3: Gated D-latch

By hooking the S and R inputs in this way, we ensure that we are only setting or resetting the latch. This will prevent the invalid input combination of (1, 1) into the SR-latch. While this is a step in the right direction, the D-latch will propagate data whenever the clock is high, which is not the same as the ability to propagate data on the rising edge of a clock. To propagate on the rising edge of a clock, one can utilize two latches and arrange them in a master-slave configuration, which is shown below:

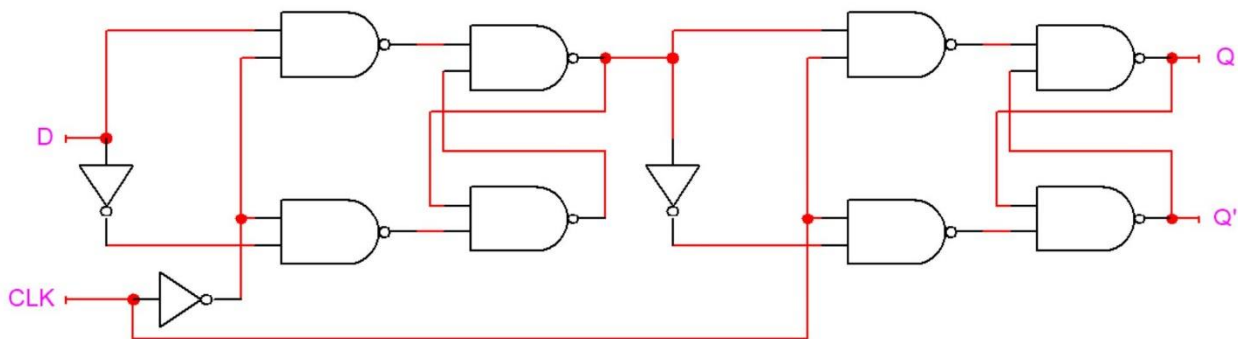


Figure 4: Master-Slave D Flip-flop

Here, what we see are two gated D-latches, where the Q output of the first (master) latch is supplied as the D input into the second (slave) latch, and the gating parameter is inverted between the two latches. The inversion of the clock is what underlies the propagation of the D input to towards the Q output on the rising edge of the clock. When the clock is low, the master latch is enabled, and sets the value according to the input D. The slave latch is disabled and maintains its previous value. At the moment when the clock goes high, the slave latch will become enabled and will change to match the value of the master latch. At the same time, the master latch becomes disabled and holds its value. Thus the slave latch will hold its value as well for the rest of the time that the clock is high (it will also hold its value when the clock goes low again because it will subsequently become disabled). So, the output Q only changes during the transition of the clock from 0 to 1. A more detailed timing analysis of the master-slave D flip-flop can be found in chapter 11 of your textbook.

Sequential Circuit Design Flow

Now that we have established the ability to construct flip flops out of simple combinational gates, one can develop some theory pertaining to the design of sequential circuits. Sequential circuits require state information, in addition to inputs, to generate their outputs. Accordingly, the key elements of sequential circuits are flip-flops, which are used to hold state information. The output signals and next

state values can each be calculated by combining flip flop states and circuit inputs using combinational logic.

The conventional way to compute the combinational logic functions for next-states and outputs is through a state machine representation. The problem statement is first interpreted into state diagrams that demonstrate the transitions between states of the circuit and the input/output conditions. State diagrams, or state machines, not only resemble flow charts in principle, but also have to include timing and state characteristics for the sequence of events. State diagrams establish the starting point for the design of sequential circuits. The entire design procedure of sequential circuits can be summarized as follows:

1. Obtaining the state diagram for the sequential circuit to be designed.
2. Minimizing states if possible.
3. Assigning binary codes to the states according to the encoding algorithm chosen.
4. Generating next state and output tables based on the state diagram and the excitation table of the selected flip-flop.
5. Deriving the logic functions for next-states and outputs.
6. Implementing the logic with flip-flops and combinational gates, as specified by the equations.

Mealy and Moore Machines

One implementation detail concerning finite state machines involves the parameters that should be utilized when evaluating the output. A *Mealy machine* is an FSM in which outputs are generated based on the current state as well as the current input. A *Moore machine*, on the other hand, generates outputs based on the current state only. Examples of these two types of FSMs can be found in Chapter 13 of the textbook.

State Minimization

The minimization of states is based on the observation that equivalent states can be merged. How does one identify if two states are equivalent? A cursory examination will indicate that satisfaction of two straightforward conditions will suffice in determining equivalence. First, the two states in question must produce the same output on every outgoing transition. Second, the two states must transition to the same states when given the same inputs. While these two conditions suffice, one can observe that the second condition can be relaxed, thus expanding the possibilities for equivalence. Transitioning to the identical state (on the same inputs) suffices, yet transitioning to an *equivalent* state (on the same inputs) will do as well (since if the states are equivalent, they will reduce to the same state, thus making this expanded condition equivalent to the original one). In other words, if the two states transition to *different, but equivalent* states when given the same inputs, the two states will still be equivalent (as long as they fulfill the first constraint as well).

For example, let's say we have two states, S1 and S2 that transition to the exact same states on identical inputs (and produce identical outputs on the same inputs). This leads us to conclude that S1 and S2 are equivalent. Now, suppose we have two more states, S3 and S4. For a particular input, S3 transitions to S1; similarly, when given the same input, S4 transitions to S2. All other outputs and transitions are identical between S3 and S4. Although the destination of the transitions for S3 and S4 are different for that particular input, the states S3 and S4 are still equivalent because S1 and S2 are equivalent. This formulation can be rewritten as an implication, with the equivalence of S3 and S4 implying the equivalence of S1 and S2. Conversely, the nonequivalence of S1 and S2 would imply the nonequivalence of S3 and S4. This gives rise to an algorithm for state minimization that utilizes an **implication table**, an example of which can be found in Chapter 15 of your textbook.

Encoding Heuristics

In order to assign binary values to the states, state encoding strategies must be employed. In this lab, you will be concerned with the **minimum-bit-change heuristic** and the **prioritized adjacency heuristic**. The prioritized adjacency encoding strategy is discussed in more detail in Section 15.8 of your textbook as well.

Minimum-Bit-Change Heuristic

The **minimum-bit-change heuristic** suggests the assignment of binary encodings to states so as to minimize the total number of bit transitions between all states. The method can be applied by assigning weights to edges in the diagram such that each weight represents the number of bit-flips between two neighboring states multiplied by the number of input combinations causing the transition. The algorithm aims to find the minimum sum of weights, that is, the state encoding satisfying the minimum number of bit transitions while traversing the diagram.

The minimum bit heuristic can be helpful for SR flip-flops, but not necessarily of use for JK flip flops. The minimum bit heuristic attempts to introduce extra don't cares in the next-state K-maps. In the case of SR flip-flops, you will remember that remaining at the same state can be expressed by an SR utterance that has one don't care (i.e. $0 \rightarrow 0$ is $SR = 0X$, and $1 \rightarrow 1$ is $SR = X0$), while toggling your flip-flop values needs an SR utterance with no don't cares whatsoever. In the case of JK, however, the utterances are identical to SR for staying at the same value but an additional don't care is introduced for toggles (i.e. $0 \rightarrow 1$ is $JK = 1X$ while $1 \rightarrow 0$ is $JK = X1$). Thus, in the case of JK you will always get one don't care whereas in the case of SR you will get a don't care only in the case of constant flip-flop values, which a minimum Hamming bit heuristic will maximize. (For D and T once again the minimum Hamming distance heuristic is of no use, as you will get no don't cares in any case.)

Prioritized Adjacency Heuristic

The **prioritized adjacency heuristic** assigns binary encodings that are Hamming distance 1 away (i.e. adjacent encodings) to state pairs in a prioritized order. As there exists a limited number of single bit distance codes, the prioritized adjacency heuristic starts by allocating adjacent encodings to the states with the highest priority, namely, the pairs of states that have the same next state under the same input condition. Once the highest priority state pairs have been handled, the heuristic suggests single Hamming distance assignments to state pairs that are next states of the same state as long as they are transitioned into on Hamming adjacent input combinations. Finally, state pairs that have identical output behaviors for the same inputs (an easier to satisfy constraint for a Moore implementation, as the output does not depend on inputs) are accorded single Hamming distance assignments, presuming of course that any such are still available. Each of the heuristics outlined in turn can be prioritized by a weight that reflects the number of times that it holds for that particular pair of states. These heuristics aim at placing the same values in neighboring rows and columns in the next-state and output Karnaugh maps. In this way, larger groupings of minterms (or maxterms) can be attained and more compact logic functions can be derived. As this heuristic aims at attaining topological affinity, the main principle underlying this algorithm is independent of the particular flip-flop type, and should deliver improvements to any flip-flop choice on the average case.

As an example of applying this heuristic, we will use the "101" sequence detector FSM found in Chapter 14 of your textbook. The first rule in assigning adjacent states is that states which have the same next state for a given input should be given adjacent assignments. Using the example of the "101" sequence detector, it should be evident that states (S_0 , S_1) should be adjacent, because both states transition to S_1 on an input of 1. Similarly, states (S_0 , S_2) should be adjacent, because these states both transition to S_0 on an input of 0. The second rule that should govern the decision for your state encoding is that states which are next states of the same state should be given adjacent states (as long as the input combinations that govern the transitions are Hamming distance 1 away, which is trivially satisfied for this example as there is only 1 input variable). In the example, the sets of states that satisfy this heuristic are the next states of S_0 , S_1 , and S_2 or (S_0 , S_1), (S_1 , S_2), and (S_0 , S_2),

respectively. In summary, we have the following sets that are specified by the first and second adjacency heuristics:

1. $\{ (S_0, S_1), (S_0, S_2) \}$
2. $\{ (S_0, S_1), (S_1, S_2), (S_0, S_2) \}$

We should attempt to fulfill as many of these adjacency conditions as possible, but it is worth noting that in most cases you will be unable to satisfy every constraint set by the heuristic. If this is the case you should first attempt to satisfy every set in your first adjacency heuristic and then move on to satisfying your second heuristic (and prioritize by the number-of-times weight discussed within each heuristic). By letting, S_0 equal "11", S_1 equal "01", and S_2 equal "10" we have satisfied the entire set of the first adjacency heuristic. In this case, we have also satisfied two of the sets in the second group (which you will notice is the maximum number of satisfiable second heuristic constraints). In your implementations of the sequence detectors, make sure to adhere to the same procedure illustrated for this simple example. Following this heuristic will result in minimizing the necessary logic gates for your next state logic.

III. Implementation

Part 1: Flip-flop Design

In this part, we ask that you implement your own D, T, SR, and JK flip-flops. You have seen the implementation of the D flip-flop in the previous section, and can implement it similarly. As you have probably figured out, a T flip-flop can then be trivially implemented with a D flip-flop. However, the construction of an SR flip-flop requires more care. The following is one example of an SR flip-flop implementation, which also utilizes a master-slave architecture similar to that of the D flip-flop:

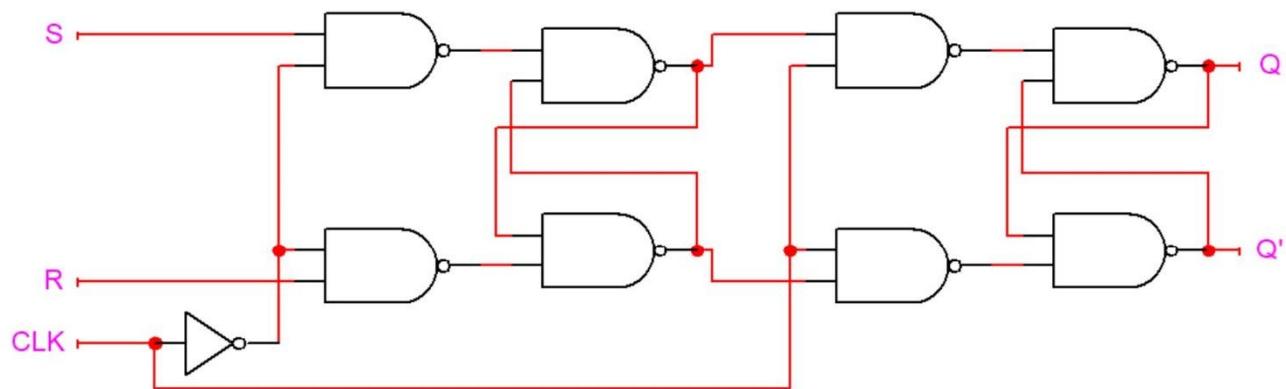


Figure 5: SR Flip-Flop

However, it should be noted that the implementation of the SR flip-flop displayed above will fail in certain corner cases. As you recall, flip-flops, by definition, propagate outputs to Q and Q' on the rising edge of the clock. However, due to the ability to set S and R separately, it is possible to induce an unwanted state change in the master latch that can propagate to the slave latch. For example, let's say we reset the flip-flop by supplying $S = 0$ and $R = 1$ to the inputs of the flip-flop. When we toggle CLK so that it is 1, we will induce a reset, making $Q = 0$ and $Q' = 1$. Now, let's do the following:

1. Set CLK to 0
2. Set $S = 1$ and $R = 0$
3. Set $S = 0$ and $R = 0$
4. Set CLK to 1

According to the specification of an SR flip-flop, the resulting outputs should retain the value we discussed in the previous paragraph of $Q = 0$ and $Q' = 1$ because, on the rising edge of CLK (which is when the flip flop should be sensitive to the input combinations), S and R should both register 0. Unfortunately, because of the master-slave architecture, when CLK is 0, setting S to 1 and R to 0 will change the state of the master latch, which propagates on the rising edge of CLK to the slave latch to yield an incorrect output of $Q = 1$ and $Q' = 0$.

How does one fix this corner case? As you have probably realized, the problem arises due to the ability to change the state of the master latch before the rising edge of the clock. While changing the state of the master latch is desirable when setting or resetting the flip-flop, one must be sure to not let the state of the master latch propagate to the slave latch when S and R are both set to 0. Ensuring this functionality is key to the success of a correctly functioning SR flip-flop. The JK flip-flop can be constructed from the correctly functioning SR flip-flop.

Once you finish pondering and correcting this timing behavior (which should be a fairly small modification), please proceed to augment your implementation of these four flip-flops so that they possess asynchronous set and reset inputs. The concept of asynchronous inputs should be familiar, as you have used them in your previous labs. However, as a reminder, an asynchronous reset is a signal that, when asserted to 1, will set Q to 0 and Q' to 1 irrespective of the clock. Likewise, an asynchronous set is a signal that, when asserted to 1, will set Q to 1 and Q' to 0 irrespective of the clock. It should be noted that having both the set and reset be 1 constitutes an illegal entry which you shouldn't encounter, and can therefore be implemented however you see fit.

How does one construct an asynchronous set and reset? As you recall, the fundamental building block of a flip-flop was the SR-latch, and converting the SR-latch into a flip-flop required numerous gates to be added on top of the latch to ensure synchronous behavior. An asynchronous set or reset should thus possess the ability to bypass all of the synchronous aspects of the flip-flop so that they set or reset the latches directly.

The requirements for this part of the lab are reiterated below:

- You are to implement four types of flip-flops: D, T, SR, and JK
 - The T flip-flop is based on the D flip-flop
 - The JK flip-flop is based on the SR flip-flop
- All flip flops are sensitive to inputs only on the rising edge of the clock (i.e. the SR (and JK) flip-flop must address the described corner case)
- All four flip flops will have asynchronous set and reset inputs

For each flip-flop design, we ask that you create a LogicWorks component, for a total of four *.cct files. You may use the gates found in the standard libraries in LogicWorks in your implementations. There is no need to modify the default delays since we are not concerned with the propagation delays for this assignment.

Part 2: Vending Machine FSM Design

You have been tasked with designing the controller for a vending machine that dispenses candy. It is the year 2030, and with the increasing support of cryptocurrencies, the value of the US dollar has plummeted to a record low. Still, the government cannot formally back such currencies, and companies must use the far inflated US dollar to operate within legal bounds. Furthermore, this

particular vending machine will be deployed in Disneyland, making the price of a single chocolate bar even more exorbitant. A single candy bar will cost \$15.00.

As you know, FSMs can be implemented in a Mealy or in a Moore fashion. This vending machine will follow the Mealy model. It will receive as input either a \$5, \$10, or \$20 dollar bill, or alternatively a refund signal to return the deposited bills (in case you change your mind about the nutritional benefit of candy). Candy in this vending machine will be automatically provided when the user has inserted at least \$15, at which point any remaining change will be immediately returned. If the user has put in less than the requisite \$15 to receive candy, he can choose to have his money returned by providing the reset input. The inputs are encoded as follows:

- 00: Refund signal
- 01: A \$5 dollar bill has been inserted
- 10: A \$10 dollar bill has been inserted
- 11: A \$20 dollar bill has been inserted

For this vending machine, there are two sets of outputs to consider. The first is whether to provide a user with a candy or not (1: candy, 0: nothing). The next output to consider is how much change to return to the user. Considering that the smallest bill the user can insert is \$5 (\$1 dollar bills having been taken out of circulation in 2028 due to their worthlessness in the current economy), and that candy costs \$15, we can determine that the amount of change returned by the machine may only be \$5, \$10, or \$15. We can thus provide the output of the vending machine change with the following encoding scheme:

- 00: No change
- 01: Return \$5
- 10: Return \$10
- 11: Return \$15 (a \$5 dollar bill and a \$10 dollar bill)

Now that we have defined the inputs, outputs, and overall behavior of our vending machine, let us discuss the state diagrams for implementing this as a Mealy machine.

Mealy-based Vending Machine FSM Design

When considering the vending machine design, we can consider the present state in the Mealy-based FSM to denote the amount of change stored in the machine. In the Mealy machine, the outputs are determined by the current state and the current input. There are four different possible input values (shown in the following figure on top), and two types of outputs (shown under the bar below the input, and separated by a ' | '). The first signals whether the vending machine user should receive candy. The next two output bits show (in the encoding form just defined) the amount of change the user should receive.

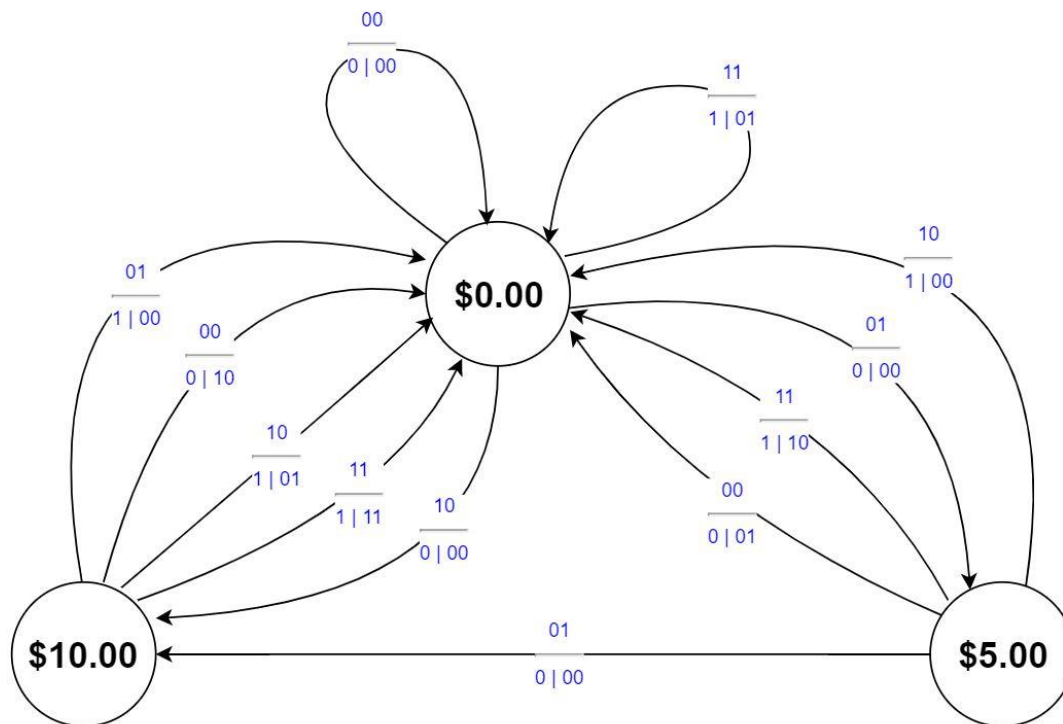


Figure 6: The three-state Mealy FSM for the vending machine

After looking at Figure 6 for the Mealy state machine, you may notice that there is not much you can do in terms of minimizing this FSM. You will need to implement this state machine using **behavioral Verilog**. You will need two flip flops to denote all four states.

Implementation details

You are to implement the design for the vending machine in Verilog. It will follow exactly the design and behavior shown in Figure 6. A transition to the next state is triggered by the rising edge of the clock, which your Verilog module will take as input. This module that you write will then be used by a *super-controller* module that we will provide to handle some of the timing semantics, as there are some timing issues that need to be considered when implementing this Mealy machine:

- The Mealy machine will transition states whenever there is a rising edge of the clock. However, if no money is inserted and the refund signal is not pressed before the next rising edge of the clock, then there shouldn't be a state transition. Instead, the super-controller will only input a rising clock edge into your Mealy machine when there is an output.
- If the Mealy machine is waiting on a particular state, and the inputs indicate that candy and/or change should be output, it will continue to have that output until either the inputs change or the rising edge of the clock signal is received. The super-controller will ensure that the candy and/or change is only output once (for one clock cycle).
- The super-controller will have an additional input, which will be a trigger that indicates that a new input was received by the vending machine. The super controller will use this input to trigger the clock of your Mealy machine, as well as to make sure that only one candy bar and one set of change will be output.

Although Verilog is a very powerful tool that will handle much of the next-state and output logic for you, you will still need to provide encodings to be used for each state. Please use the **prioritized-adjacency heuristic** to determine your state encodings.

Please do not modify the module definition for either the super-controller or for the Mealy machine

module, as they will need to be kept the same to work together and with the testbench. You will not have to make any changes to the super-controller module. A testbench will be provided, though you will need to add additional test cases to ensure correctness of your design.

Part 3: Booth's Multiplication with FSMs

Our Booth's multiplier from Lab 2 had an obvious shortcoming. Even though we could "skip" add operations during long strings of 1's or 0's in our multiplier, our circuit still performed the operation! (Though, the result of the operation was discarded on cycles where the add/sub wasn't needed.) The natural next step is to provide functionality to skip unnecessary add and subtract operations. With your new expertise in finite state machines, it is now possible to design a new controller that can skip those operations!

We ask that you implement the new controller twice: once as a Mealy machine and once as a Moore.

3A: Mealy implementation of Booth's controller

When performing Booth's multiplication, we scan the multiplier from right to left by examining two bits in every step. Therefore, there are a total of 4 possible patterns in the bits we are examining, each of which corresponds to an arithmetic operation (addition, subtraction, or no-op) specified by the Booth's table. From a sequential circuit design perspective, we can also view these four patterns as four different states of an FSM. The transition from one state to another state will be triggered by the right shifting of the multiplier, as the new bit that is shifted into the two-bit window changes the bit pattern that is under examination. The arithmetic operation that needs to be performed in each step can just be determined by the current state and the new bit that is shifted in.

Based on these concepts a four state Mealy FSM for the Booth's algorithm can be constructed as in Figure 7.

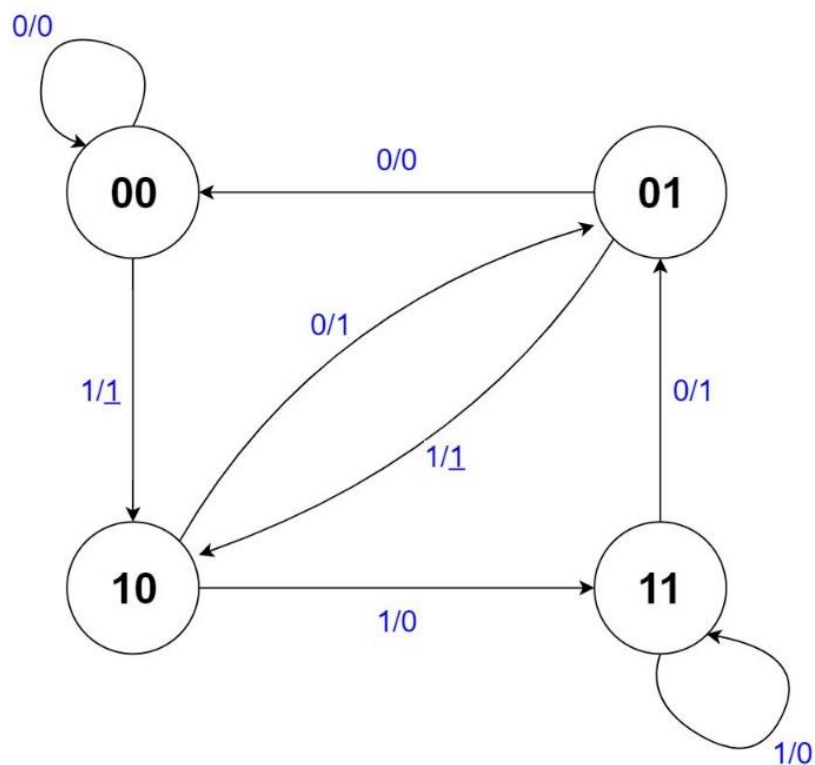


Figure 7: The four state Mealy FSM for Booth's

As you can see, each edge is annotated with a pair of values separated by a slash("/"). The left symbol on an edge denotes the input value of the FSM. The value after the slash is the output value, where a 1 indicates an addition, a 1 indicates a subtraction, and a 0 indicates a no-op. You may wonder how a single state can have two different outputs for the same 2-bit pattern in Booth's algorithm. For example, we have usually associated 00 with a no-op, according to Booth's algorithm as we saw it in Lab 2, but here the output of the 00 state can be either a 0 or a 1 (either a no-op or a subtraction). The puzzle can be resolved by realizing that the pattern associated with each state in the Mealy machine represents the *previous* pattern held in the 2-bit window under examination in Booth's, and the input to the FSM is the bit that was newly shifted into the 2-bit window (that is, the FSM input is the current least significant bit of the multiplier). Since the previous value of the 2-bit window is remembered by the current state, we no longer need the extra flip-flop after the least significant bit of the multiplier to hold the last value shifted out of the multiplier in our Booth's circuit from Lab 2. An additional benefit of this representation is that since the state is associated with the previous value, the Mealy machine can be directly initialized to state 00 (even if the LSB of the original multiplier is 1).

As you might have already noticed, the four-state FSM above can be optimized as some states within this diagram are equivalent. The current diagram requires 2 flip-flops to maintain the current state. But, after performing state minimization, you will be able to reduce the number of states to just 2! The state machine must also have a 2-bit output, as there are three possible outputs (add, subtract, or no-op).

So now we have a Mealy machine that can specify which operation to perform. The next question is how we can use this new FSM to control the whole circuit and skip operations. As you recall, the Lab 2 implementation contained a separate *controller* component that controlled whether to shift or perform an add/subtract operation. Besides the *init* and *end* phases, the controller simply ticked back and forth between add/subtract and shift phases. Our current Mealy machine outputs whether to add, subtract, or no-op (do nothing) during an add/subtract phase. First, we can have the Mealy machine make a state transition according to the clock signal, instead of whenever a shift is performed. Then, if we plan to skip the add/subtract phase whenever the Mealy machine outputs a 0 (a no-op), we can instead have that output specify to perform a shift operation, effectively skipping the no-op. If the Mealy machine outputs a 1 or a 1, then the circuit will use that as the control signal to enter the add/subtract phase, as well as whether the multiplicand should be added or subtracted.

We are well on our way to getting a properly functioning controller. The only problem is that we must also perform a shift after every add or subtract operation, which we cannot perform at the same time as the add/subtract operation itself. However, this turns out to be a non-issue. The Mealy machine will transition states after an add/subtract operation, since it transitions states according to the clock signal. The input to the FSM, however, will not change during this transition, since no shift has occurred. But, if you examine the FSM and consider the case for addition, every input that specifies an output of 1 (for add) on a particular transition will cause a transition to a state where, given the same input, the output is 0 (for shift) and that transition points to the same state. Thus, we are guaranteed a shift after every add operation! In the same manner, we are also guaranteed a shift after every subtract operation.

There is one final piece that must be addressed. What about the *init* and *end* phases? We can easily add two more states back to our FSM, one for *init* and one for *end*. When the circuit is reset, the FSM will be reset to the *init* state. Because the states in Figure 7 were representative of the *previous* value of the 2-bit window in Booth's, after initializing the circuit, we can always start from the 00 state. Then the *init* state will have a single transition to the 00 state (or whatever the equivalent state is after state minimization) that outputs an initialization signal. Now we have 4 outputs: 0, 1, 1, and 'initialize'. Since we still have 4 or less outputs, we can keep the FSM output to just 2 bits. The transitions to the *end* state are a bit more nuanced. The original controller from Lab 2 would transition to the *end* phase after N shift phases, for N bit multiplication. Our states would need to keep track of how far along the algorithm we are, implying that we need at least N states that cannot be minimized! This is obviously

less than ideal. Instead, we will employ a counter that will keep track. Whenever a shift occurs, the counter will be signaled to increment by 1. The FSM will have an additional input that will signal when the counter has counted to N, indicating that N shifts have occurred. This input will cause the 00 and 11 states (or whatever the equivalent states are) to transition to the *end* state, where the FSM will remain until the circuit is reset.

Finally, we have a complete controller that can both control the datapath by indicating whether a shift or an arithmetic operation should be performed, as the original controller from Lab 2 did, as well as indicate whether an add or a shift should be performed, all while skipping unnecessary operations! The procedure to designing this newer, smarter controller is summarized below:

1. Perform state minimization on the Mealy FSM shown in Figure 7. You should be able to reduce it to just 2 states.
2. The FSM state transitions will be triggered by a clock input. The input is the least significant bit of the multiplier register.
3. Instead of outputting 'add', 'subtract', and 'no-op', the FSM should output 'add', 'subtract', and 'shift'.
4. Add an *init* state. The *init* state should transition to either state '00' (or the equivalent state), and output an initialization signal.
5. Add an *end* state, and an external counter to keep track of the number of shifts. States '00' and '11' (or the equivalent states) should transition to the *end* state when the counter reaches N, where N is the width of the operands of the multiplier. The end state should output a logic 1 on a special output of the controller, signifying the circuit has completed the multiplication.
6. Because the new controller specifies both whether to perform a shift or an arithmetic operation, as well as whether the arithmetic operation should be an addition or a subtraction, the datapath from your Lab 2 multiplier must be simplified to use the new signals.

Implementation Details

The state names (shown as 00, 01, 10, and 11 in Figure 7) do not necessarily match the encodings of your states. After state minimization, you will only have 2 states left before adding 2 more states (for *init* and *end*), for a total of 4 states. You may encode these four states however you wish, keeping in mind that the majority of transitions will be between the original two states (not the *init* and *end* states).

The 2-bit output of your Mealy FSM should have follow the following scheme:

- **00**: Initialize – This output is only used when transitioning from the *init* state.
- **01**: Add operation
- **10**: Subtract operation
- **11**: Shift operation

There should be one more 1-bit output to signal the completion of the multiplication:

- **Done**: A logic 1 on this output signifies the circuit has completed the multiplication. Else, this should be logic 0.

There are 4 1-bit inputs to the FSM:

- **Shift-In**: The last bit shifted into the 2-bit window examined by Booth's algorithm (i.e. the LSB of the Multiplier register).
- **Completed**: A logic 1 on this input means N shifts have occurred, and a transition to the End state should occur next. Else, the input should be a logic 0. This input is driven by a counter (and a few extra gates).
- **Clock**: The clock signal for the circuit.
- **Reset**: Synchronous Reset. A logic 1 on this input should reset the FSM to the *init* state.

Use the state encodings, inputs, and outputs to derive the next-state logic and output logic for the new Booth's controller. Only the **Shift-In** and **Completed** inputs need to be considered when deriving the

logic. Use the **minimum-bit-change heuristic** when determining your state encodings for the 4 states. Please implement the controller using **SR flip-flops**. Because the new controller will handle both the current state of the machine, as well as which operation the Adder/Subtractor component will perform, you will need to modify the datapath of the Booth's multiplier to use the FSM outputs appropriately (you will mostly just need to remove the old logic that determined whether an add or a subtraction was necessary).

We ask that you implement the new controller in LogicWorks as a new component, and place it into your Booth's circuit from Lab 2. We have already provided a solution for Lab 2 when Lab 3 was assigned. You may download and reuse that solution as the basis for this lab, if needed.

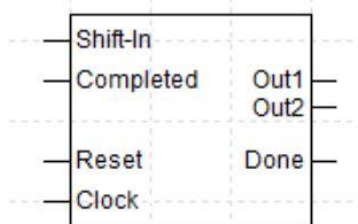


Figure 8: The Mealy implementation of the Controller Component

3B: Moore implementation of Booth's controller

Luckily, moving to the Moore implementation does not deviate significantly from the process of creating the Mealy machine. Similar to the Mealy representation, we start with a 4-state Moore machine based on the original Booth's algorithm.

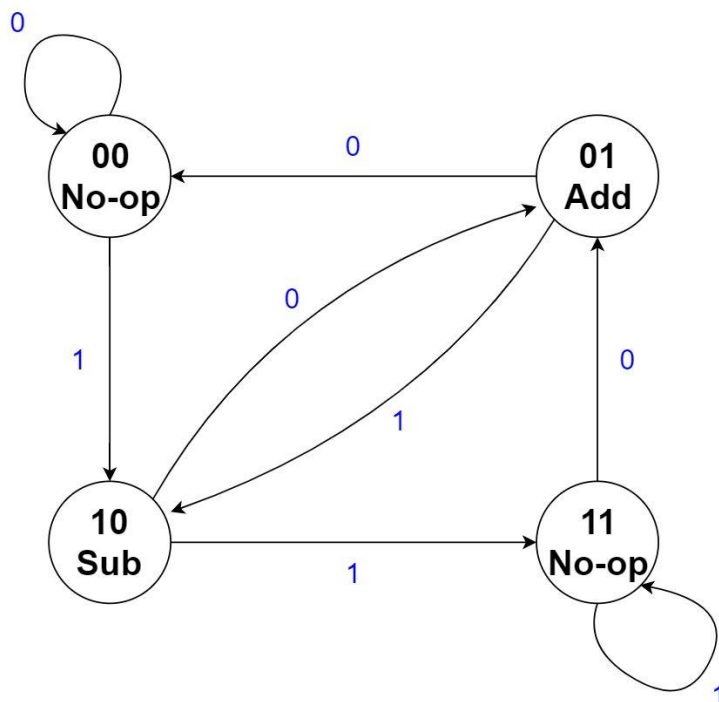


Figure 9: The four state Moore FSM for Booth's

Now, the outputs are associated with the current state, rather than with the current state and the input together. Unfortunately, this means that no state minimization can be performed since the only two states with the same outputs are states 00 and 11, which clearly do not have equivalent transitions.

An important difference between this Moore machine and the Mealy machine in Figure 7 is where the input comes from. In the Mealy machine, it came from the least significant bit of the multiplier, which is

inside of the 2-bit window that is examined by the traditional Booth's algorithm. This was not a problem because the output was in part determined by that input. However, in the Moore machine, the input cannot affect the output, but only which transition to take to the next state. If the two-bit window contains 01, for example, then the current operation should be an Add operation and the FSM should be in the 01 state. Thus, the input to the Moore FSM will be the next bit that *will* be shifted in to the two-bit window (rather than the bit that was *already* shifted in for the Mealy FSM). More concretely, the input to this FSM is the *second* least significant bit of the multiplier.

Employing a similar strategy as we did for the Mealy machine, we can have the no-op output instead be a 'shift' output, and have the FSM make a transition based on the clock. However, we still need to somehow ensure a shift occurs after every add or subtract operation. This naturally worked out nicely with the Mealy FSM, but closer inspection of the Moore FSM reveals that we are not quite so lucky. For example, if we are in state 00 and the next bit to be shifted in is a 1, then we will shift in the 1 and make a transition to state 10. The next input could be either a 1 or a 0. Depending on the input, this results in two scenarios:

- If it is a 1, then we transition next to the 11 state (after performing the subtraction), and since no shift occurred, the input is still 1. We then perform a shift and transition to the same state. In this case, everything works out perfectly, as it did with the Mealy FSM.
- If it is a 0, then we next transition to state 01. So, immediately following the subtraction operation is an addition (that, in the absence of a shift, will cancel the two operations out). Since no shift has occurred yet, the input is still 0 and we transition to the 00 state, where we finally get a shift. So, the machine has effectively 'skipped' a non-zero bit in the multiplier, and the partial product is now incorrect.

In conclusion, the first scenario works out, but in the second scenario, we still need a shift to occur between the subtract and the add. You may notice, however, that although the 10 and 11 states are not equivalent because the outputs are nonequivalent, the transitions *are* equivalent. If we 'force' the FSM to transition from 10 to 11 (after the subtraction), then we will guarantee that a shift will take place. And, since the transitions are equivalent, we will end up going to the correct state on the next cycle anyway! Thus, we can simply remove the transition from state 10 to 01, and have the FSM transition from 10 to 11 regardless of the input. We will do something similar for the transitions from the 01 state. Then, everything will work out correctly, and we will still be able to skip unnecessary operations! The following is the new, updated Moore state machine, where a '?' indicates a 'don't care' input:

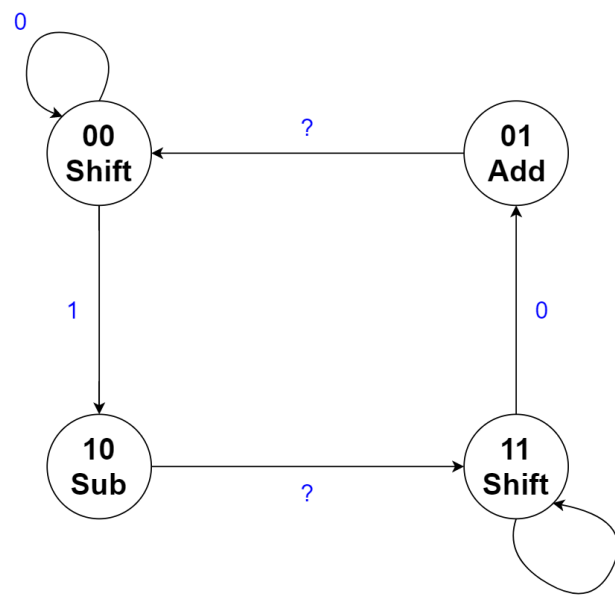


Figure 10: The modified four state Moore FSM for Booth's

Similar to the Mealy machine, we just need to add an additional *init* and *end* state. The 00 and 11 states can transition to the *end* state, governed by a counter that counts the number of shifts that have occurred. However, there is one more hurdle with the Moore implementation. The *init* state must transition to either the 00 or the 10 state, depending on the least significant bit of the multiplier. Thus, the LSB of the multiplier must also be an input to the Moore FSM, used only to determine the transition from the *init* state.

Implementation Details

If we consider the 4 states required as shown in Figure 10, as well as the additional 2 states for *init* and *end*, then we will need 3 flip-flops to store the current state. Additionally, we will have a 1-bit input for the next bit to be shifted into the 2-bit window examined by Booth's, a 1-bit input to signify when we have completed N shifts, and a 1-bit input to signify whether the *init* state should transition to state 00 or 10, for a total of 3 bits of inputs. This will result in a 6-variable K-map, which would be a nightmare to use! Instead, for simplicity, you may ignore the *init* and *end* states, as well as the input that signifies which state to transition to after the *init* state, when deriving your next state logic and output logic. Then, after adding in the *init* and *end* states, you may use whatever encoding you wish. This should be easy enough to do, as the input we ignored is only used once by the *init* state, and the logic to transition to the *end* state should be fairly separate and straightforward. Your Moore implementation of the new controller will have the same outputs as in the Mealy Machine. The inputs are summarized below:

- **Shift-In:** The next bit to be shifted into the 2-bit window examined by Booth's algorithm (i.e. the second to least significant bit of the Multiplier register).
- **Completed:** A logic 1 on this input means N shifts have occurred, and a transition to the End state should occur next. Else, the input should be a logic 0. This input is driven by a counter (and a few extra gates).
- **Current-Shift:** If logic 1, then the *init* state should transition to the '10' state. If logic 0, then the *init* state should transition to the '00' state. This input should be driven by the LSB of the multiplier register.
- **Clock:** The clock signal for the circuit.
- **Reset:** Synchronous reset. A logic 1 on this input should reset the FSM to the *init* state.

Again, the Clock and Reset inputs are not used when deriving your next state and output logic. Additionally, the Current-Shift input can be ignored when deriving your logic for simplicity. As with the Mealy version, please use the **minimum-bit-change heuristic** when determining the 4 state encodings, and implement the design with **SR flip-flops**. For encoding of the *init* and *end* states, you may encode and implement them in any manner you wish.

We ask you to use this controller integrated into your Lab 2 Booth's multiplier, as you did for the Mealy machine. As the outputs match the Mealy controller, changing the Mealy controller to the Moore controller should be an easy task.

Part 4: Sequence Detectors Galore

The first time you saw your new boss, you knew your stars were not aligned. Yet confident of your long seniority in what used to be the shining start-up of Wall Street a decade ago, you knew that your job was secure. Shielded off from the real world, apparently your naiveté had no bounds. Soon you realize that your boss, who for some inexplicable reason has an axe to grind, is set on giving you impossible assignments, trying to generate a paper trail against you in preparation for firing you.

Your first assignment consists of giving you a 5 state Finite State Machine (FSM) with two outputs Z_1 and Z_2 . Each of these outputs generates 1 upon seeing a 3-bit input. Immediately you realize your boss' strategy upon being told that you will not be given the 3-bit sequences activating Z_1 and Z_2 . No

worries, you say, since you are an expert in setting up recognizer FSMs from your study of Chapter 14 of your textbook. Not only do you know how to set them up, but you can also by now decode them and figure out which sequences they are detecting.

The magnitude of your task becomes clearer when you are told that the input sequences that activate the transitions will not be given to you either. Only the output behavior and the FSM! In desperation you steal a furtive glance at the notes of your new boss' right-hand man, who seems to have the complete information, and manage to see two input conditions on the transitions on this Mealy machine. You are a bit happy that there seems to be a single input variable.

While you are busy contemplating your fate, trying to figure out how to implement this FSM with unknown inputs on transitions, trying to act like a team player while ruing the fact that you should have invested more time reading murder mysteries, you're awakened out of your deep thoughts by hearing that your task has increased in complexity, and is actually to implement an FSM that initially recognizes only Z_2 , and then recognizes Z_1 as well once the first instance of Z_2 has been recognized (it will recognize both Z_1 and Z_2 after the first time Z_2 is detected). And, you are supposed to implement all this in minimal form! You vaguely recall studying a similar FSM in your Chapter 14.

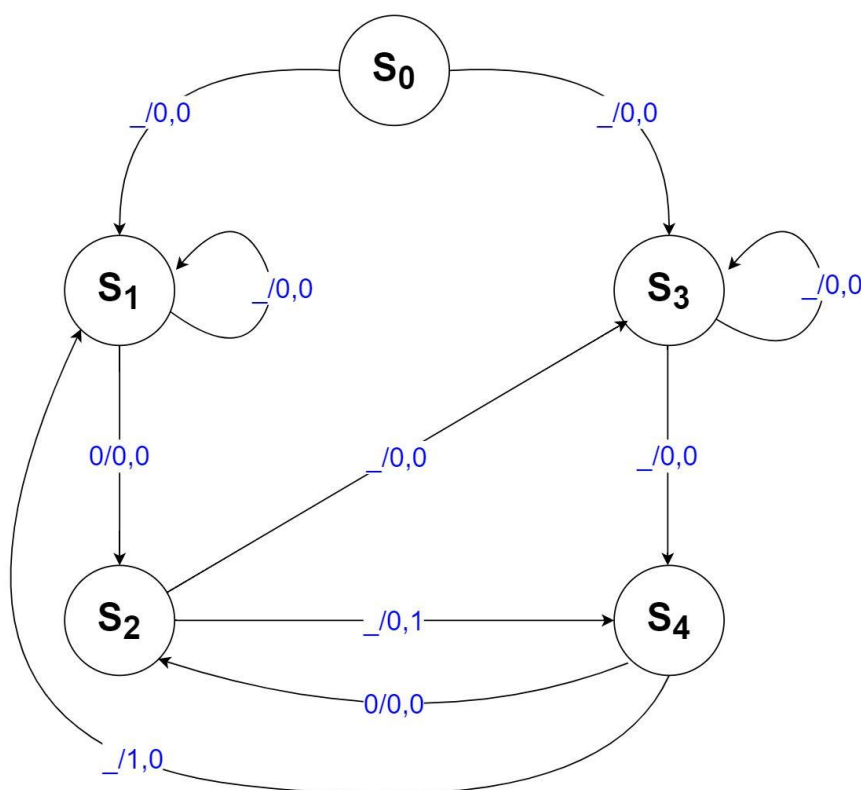


Figure 11: Original Mealy State Machine to detect Z_1 and Z_2

In an emotional state laden with anger, denial, and grief mixed in with technical arrogance (and once you are past the stage of reading murder mysteries) you decide to invest in a rereading of Chapter 14 of your textbook. You soon realize that the analogous example in your textbook is a classic example of FSM composition, wherein once Z_2 has been detected once, a reduced form of a second FSM (one that only recognized Z_2 henceforth) will be transitioned into. You do have a partial FSM (that recognizes Z_1 and Z_2) but you are at your wits' end as to how to construct the second reduced FSM (the one that recognizes Z_2 only).

Being the top student that you were in your long-gone days of youth when you were taking CSE 140(L) at your alma mater, and remembering your prowess at coming up with creative solutions to challenging logic design problems, you set yourself down to study your technical predicament. A bit of reflection informs you that the second reduced FSM can be simply obtained by using the first FSM and setting the Z_1 output to 0 in the cases when it was 1 in the original FSM. So, duplicating the FSM with a small change in the Z_1 output has resulted in the desired FSM. The connecting transition between the two FSMs is the Z_2 recognizing transition in the new, reduced FSM, which should go to the shadow counterpart state in the original FSM (thus connecting the two FSMs together).

Your pride at your technical prowess is unbounded and hope glimmers that you will be able, perhaps, to not only fulfill the task but one-up your boss as well. Yet your dreams take a break when you realize the next task that needs to be accomplished. How do you minimize the states in this composite FSM whose transition inputs are unknown!? You are staring at a ten-state machine and you know that unless you reduce the number of states to eight or below, your efforts will not make the cut.

You go back to review your “state minimization” knowledge and you are overjoyed at having stolen that furtive glance at your right-hand man’s scribbles of the full FSM. That little glance seems to save the day, and you are able to induce the equivalence of two states in the initial FSM (that detects only Z_2), thus saving you one.

Now that you are on a roll, you realize that to implement the FSM, you need to crack the puzzle of what the input transitions are. For this, you go back to your well-worn pages of chapter 14 of your book and realize that pattern recognizer FSM structures may provide insights into the transition inputs. You manage to fully uncover all the transition inputs on your composite FSM. With all the transitions visible, you notice that there is another pair of equivalent states that can be merged, reducing your states from the original ten down to eight and effectively saving you a flip-flop! On a roll and with a glass of wine at hand as a reward for your technical prowess, you take it a step further and notice yet another state that can be removed due to inaccessibility, rather than the usual state minimization techniques you have used. You are left with a reduced FSM that contains a mere 7 states.

Once you have gotten a handle on your boss and evened the playing field, you set on making your boss come up short by shaking up her throne and getting her fired while getting your long deserved promotion in the process. You remember overhearing her saying that the next state logic is composed of about sixteen two-input gates (less if using more than two-input gates) when using the **Sum-of-Products form** in a **D flip-flop** implementation. You decide to one-up her and use your great skills at minimum state encoding such as the prioritized adjacency coding you learned in 140(L). Seeing that you seem to have done the impossible, your long-time colleagues that had banded over to your boss’s side, sensing that you are history, are now slowly coming back to your side. One of them whispers softly something about the first flip-flop’s next state logic needing no gates whatsoever. Buoyed by your technical prowess, sensing the battlefield shifting in your direction, you set out to find the minimized next state logic. Given the sensitivity of your strategy, you consult with your CSE 140L professor of days long gone and show him your design.

IV. Deliverables

Please submit your relevant *Logicworks* and *Verilog* files. Please also submit a report. The specific deliverables for each part are described below.

Part 1:

Please submit all of your LogicWorks .cct files containing your D, T, SR, and JK flip-flop implementations. These flip-flops should possess correctly functioning asynchronous set and reset inputs, and if both asynchronous inputs are off, the flip-flops should function synchronously. The SR and JK flip-flops must also work correctly for all corner cases.

In your report, please include a description of how you implemented each flip-flop, including how you transformed the D flip-flop into a T flip-flop and the SR flip-flop into a JK flip-flop. Describe how you solved the corner case for the SR (and JK) flip-flop. Additionally, you should explain how you implemented the asynchronous set and reset functionality.

Part 2:

Include two *.v files, one for your Mealy vending machine and for the testbench. As the super-controller should be unchanged, there is no need to submit it.

In your report, show how you implemented the state encoding heuristics.

Part 3:

Include a total of four *.cct files. One file should contain the implementation of the controller as a Mealy machine with **SR flip-flops**, and a second file should contain the implementation as Moore machine with **SR flip-flops**. The last two files should contain the Lab 2 multiplier, one with the Mealy controller integrated into it and one with the Moore controller integrated.

In your report, briefly explain the datapath changes you had to make from the Lab 2 circuit, and if there were any datapath differences between the Moore and Mealy implementation when integrated into the Lab 2 circuit.

Part 4:

Include one .cct file that contains your implementation of the full FSM.

In your report, outline:

- a) The composite FSM.
- b) The composite, state-minimized FSM; show your state minimization approach.
- c) The reduced FSM with the transition inputs deciphered. Describe how you managed to resolve this puzzle and the intermediate logical inferences you resolved.
- d) The priority 1 and priority 2 groups for your prioritized adjacency technique.
- e) Describe the approach you took for resolving these groups and how you arrived at a solution that satisfies primarily priority 1 groups and secondarily priority 2 groups.
- f) Provide a table of your symbolic states and corresponding encodings.
- g) Show your next-state and output K-maps for the D flip-flops you are using.
- h) Show your logic in **SOP** form (and hopefully how you managed to have the first flip-flop next state logic to have no gates). Tell us the total number of 2-input AND/OR gates your design represents.